На предыдущем занятии мы с вами объявили два класса: базовый GeomBase и дочерний Line, которые я запишу в следующем виде:

```cpp
class GeomBase {
protected:
    int x0{0}, y0{0}, x1{0}, y1{0};
public:
    void set_coords(int x0, int y0, int x1, int y1)
    {
        this->x0 = x0; this->y0 = y0;
        this->x1 = x1; this->y1 = y1;
    }

    void get_coords(int& x0, int& y0, int& x1, int& y1)
    {
        x0 = this->x0; y0 = this->y0;
        x1 = this->x1; y1 = this->y1;
    }
};

class Line : public GeomBase {
private:
    double length{0.0};
public:
    void draw()
    {
        printf("Line: %d, %d, %d, %d\n", x0, y0, x1, y1);
    }
};
```

Зададимся вопросом: в каком порядке происходит вызов конструкторов и деструкторов базового и дочернего классов? **Я напомню, что отсутствие явно объявленных конструкторов в классах не освобождает компилятор от их использования**. В этом случае он применяет предопределенные стандартные конструкторы и деструкторы. Однако чтобы увидеть порядок их вызовов, мы с вами пропишем конструкторы и деструкторы по умолчанию в базовом и дочернем классах следующим образом:

```cpp
class GeomBase {
protected:
    int x0{0}, y0{0}, x1{0}, y1{0};
public:
    GeomBase()
```

```
        { std::cout << "Base: constructor" << std::endl; }
    ~GeomBase()
        { std::cout << "Base: destructor" << std::endl; }
...
};

class Line : public GeomBase {
private:
    double length{0.0};
public:
    Line()
        { std::cout << "Line: constructor" << std::endl; }
    ~Line()
        { std::cout << "Line: destructor" << std::endl; }
...
};
```

Если теперь в функции main создать объект класса Line:

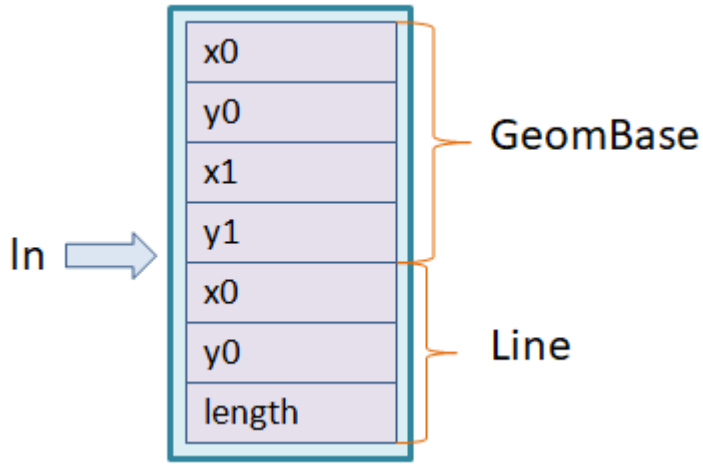
```language-cpp
int main()
{
    Line ln;

    return 0;
}
```

то при выполнении программы в консоль будут выведены строчки:

```
Base: constructor
Line: constructor
Line: destructor
Base: destructor
```

И этот порядок вполне логичен. Если мы с вами вспомним структуру объекта дочернего класса Line, то увидим вначале формирование данных базового класса GeomBase, а затем, производного от него класса Line:



Как раз за формирование первой части класса GeomBase отвечает конструктор базового класса. Именно поэтому он вызывается первым. А за формирование второй части – конструктор дочернего класса Line (он вызывается вторым). И **этот порядок всегда должен быть именно таким, иначе объект не будет сформирован корректно**.

А вот деструкторы вызываются уже в обратном порядке и это тоже логично, так как в деструкторе дочернего класса Line мы вполне можем использовать данные базового класса, поэтому они должны существовать. В результате, сначала следует освободить ресурсы, занимаемые дочерним классом и только потом – базовым классом. Это определяет порядок вызова деструкторов. **В действительности, компилятор подставляет вызов деструктора базового класса в конец тела деструктора дочернего класса**:

```language-cpp
~Line()
{
    // освобождение ресурсов дочернего класса
    ~GeomBase();
}
```

Конечно, нам явно это прописывать не нужно, компилятор все сделает за нас. А вот с конструкторами не всегда все срабатывает на автомате. Компилятор может без проблем вызывать конструктор по умолчанию базового класса, но если его не будет, то вызов любого другого ложится на плечи программиста. **Например, если в базовом классе объявить конструктор с четырьмя параметрами (вместо конструктора по умолчанию)**:

```language-cpp
  GeomBase(int a, int b, int c, int d)
        : x0(a), y0(b), x1(c), y1(d)
        { std::cout << "Base: constructor" << std::endl; }
```

то прежняя программа не скомпилируется. Компилятор не сможет самостоятельно сделать вызов такого конструктора и сформировать объект базового класса. Это мы должны будем прописать явно при вызове конструктора дочернего класса, например, следующим образом:

```language-cpp
    Line() : GeomBase(0, 0, 0, 0)
        { std::cout << "Line: constructor" << std::endl; }
```

Обратите внимание, что, так как конструктор базового класса должен вызываться перед конструктором дочернего класса, то вызов GeomBase размещен в секции инициализации. Расположить вызов в теле конструктора будет ошибкой:

```language-cpp
    Line()
    {
        GeomBase(0, 0, 0, 0);
        std::cout << "Line: constructor" << std::endl;
    }
```

В таком виде программа не скомпилируется.

Конечно, если в дочернем классе Line предполагается несколько перегруженных конструкторов, то каждый из них должен вызывать прежде конструктор базового класса GeomBase:

```language-cpp
    Line() : GeomBase(0, 0, 0, 0)
        { std::cout << "Line: constructor 1" << std::endl; }
    Line(int a, int b, int c, int d) : GeomBase(a, b, c, d)
        { std::cout << "Line: constructor 2" << std::endl; }
```

Но это только в том случае, если нет конструктора по умолчанию или нужно делегировать некоторые действия, используя другой конструктор базового класса.

Если помимо вызова конструктора базового класса нужно выполнить инициализацию каких-либо переменных создаваемого объекта, то эта инициализация записывается после конструктора базового класса:

```language-cpp
    Line() : GeomBase(0, 0, 0, 0), length {0}
        { std::cout << "Line: constructor 1" << std::endl; }
```

И это очевидно, т.к. возможно, что для формирования переменных дочернего класса потребуются какие-либо значения базового класса и они должны быть уже сформированы.